EFFICIENT BOTTOM-UP EVALUATION OF LOGIC PROGRAMS

RAGHU RAMAKRISHNAN, DIVESH SRIVASTAVA and S. SUDARSHAN
Computer Sciences Department
University of Wisconsin – Madison
Madison WI 53706, U.S.A.

Abstract. In recent years, much work has been directed towards evaluating logic programs and queries on deductive databases by using an iterative bottom-up fixpoint computation. The resulting techniques offer an attractive alternative to Prolog-style top-down evaluation in several situations. They are sound and complete for positive Horn clause programs, are well-suited to applications with large volumes of data (facts), and can support a variety of extensions to the standard logic programming paradigm.

We present the basics of database query evaluation and logic programming evaluation, and then discuss bottom-up fixpoint evaluation. We discuss an approach based upon using a program transformation ("Magic Templates") to restrict search, followed by fixpoint computation using a technique ("Semi-naive evaluation") that avoids repeated inferences. The program transformation technique focuses the fixpoint evaluation, which is a forward-chaining strategy, by propagating bindings in the goal in a manner that essentially mimics the backward-chaining behavior of top-down evaluation strategies.

After presenting the basic framework for bottom-up evaluation, we discuss several refinements that optimize the treatment of non-ground terms, improve memory utilization, reduce the cost of duplicate checking, and utilize the declarative semantics of the program to perform semantic query optimization in a number of ways. We also consider several extensions to the logic programming paradigm, and discuss how the bottom-up evaluation framework can be used to support these extensions. The extensions include support for negation, set-terms, constraint manipulation and quantitative reasoning.

Finally, we discuss several systems based upon bottom-up evaluation, including Aditi (Univ. of Melbourne), EKS-V1 (ECRC-Munich), Glue/NAIL! (Stanford Univ.) and LDL (MCC-Austin). We have developed such a system, called CORAL, and we present this in more detail.

Key words: Logic Programs, Bottom-Up Evaluation, Deductive Databases, Database Query Languages.

1. Introduction

Traditionally, logic programming language implementations have evaluated programs top-down. Recent developments in evaluation of database queries have provided a means of evaluating logic programs bottom-up while still retaining the focusing properties of top-down evaluation, based in large part on optimizing program transformations.

EXAMPLE 1.1. As a simple example of the power of the bottom-up approach, consider a definition of transitive closure

$$\begin{array}{lll} t(X,Y) & :- & e(X,W), t(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(5,Y). \end{array}$$

If e is cyclic, Prolog will not terminate on this program. Even if e is not cyclic, there are values for e such that Prolog takes time exponential in the size of the answer.

To evaluate the original query bottom-up, we first obtain the following program by first applying the Magic Templates[1] transformation (Section 3.1) and then predicate factoring (Section 5.1):

$$\begin{array}{lll} magic\_t^{bf}(W) & :- & ft(W). \\ magic\_t^{bf}(5). & & \\ ft(Y) & :- & magic\_t^{bf}(X), e(X,Y). \\ query(Y) & :- & ft(Y). \end{array}$$

Evaluating this program using Semi-naive bottom-up evaluation (Section 3.2) always terminates and computes the answer to the query in time linear in the answer size.

Essentially the same program results if Magic Templates plus factoring is applied to versions of the transitive closure expressed with different forms of the recursive rule, including the left-recursive and binary recursive forms (upon which Prolog does not terminate for any values of e). This example is presented in detail in Section 5. $\square$

This is an example where the bottom-up approach based on program transformations has notable success. Even on programs for which the transformations do not achieve such large gains, the Magic Templates transformation ensures that no "irrelevant" goals or facts are generated, and in many cases significant advantages accrue when bottom-up evaluation is used, primarily because:

- With memoing, it implements a form of dynamic programming, which eliminates a great deal of redundant computation, and
- It is sound and complete, and thus the declarative semantics of the program (which is taken to be its least Herbrand model) is preserved. This non-procedural approach allows for much more flexibility in program transformation and optimization.

In this paper, we give a tutorial introduction to the bottom-up approach to logic program evaluation. We introduce notation and preliminary definitions in Section 2, and present a summary of the bottom-up evaluation method based on the Magic Templates program transformation and Semi-naive fixpoint evaluation in Section 3. In this section, we also discuss some alternatives to the Magic Templates approach and compare top-down and bottom-up methods briefly. We present a survey of some related program transformations used in the bottom-up approach in Section 5. In Section 6, we discuss various extensions to the Horn clause logic programming

[1] Earlier versions of the algorithm were called Magic Sets. See Section 3 for details.

paradigm. We describe some systems based upon bottom-up evaluation in Section 7, and discuss one of them (CORAL) in more detail in Section 8.

We note that some related evaluation techniques (e.g., Vieille's QSQR and QoSaQ [109; 110; 111], parallel evaluation techniques (e.g., work by Gonzalez-Rubio, Bradier and Rohmer [35], Ganguly, Silberschatz and Tsur [32], Cohen and Wolfson [24], and others, e.g. [29; 75; 105; 113]), work on integrity constraints (e.g., Bry, Decker and Manthey [15], Vieille et al. [107] and others), work on production rule systems (e.g., Forgy [30], Sellis, Lin and Raschid [93], Widom et al. [112], and others, and systems such as OPS 5), work on intelligent query answering (e.g. [55; 22]) etc. are not discussed in this tutorial presentation due to lack of space. Our coverage attempts to present one coherent set of results that are indicative of the field, rather than a survey, and is doubtless influenced by our personal perspectives.

2. Notation and Preliminary Definitions

The language considered in this paper is that of Horn logic. Such a language has a countably infinite set of variables and countable sets of function and predicate symbols, these sets being mutually disjoint. It is assumed, without loss of generality, that with each function symbol f and each predicate symbol p, is associated a unique natural number n, referred to as the *arity* of the symbol; f and p are then said to be n-ary symbols. A 0-ary function symbol is referred to as a constant. A *term* in a first order language is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms. A tuple of terms is sometimes denoted simply by the use of an overbar, e.g., $\bar{t}$.

A substitution is an idempotent mapping from the set of variables of the language under consideration to the set of terms. Substitution operations are usually written in postfix form. A substitution σ is *more general* than a substitution θ if there is a substitution φ such that $\theta = \sigma[\varphi]$. Substitutions are denoted by lower case Greek letters θ, σ, ϕ, etc. Two terms t_1 and t_2 are said to be *unifiable* if there is a substitution σ such that $t_1[\sigma] = t_2[\sigma]$. σ is said to be a *unifier* of t_1 and t_2. Note that if two terms have a unifier, they have a most general unifier that is unique up to renaming of variables.

A clause is the disjunction of a finite number of literals, and is said to be Horn if it has at most one positive literal. A Horn clause with exactly one positive literal is referred to as a *definite clause*. The positive literal in a definite clause is its *head*, and the remaining literals, if any, constitute its *body*. A predicate definition consists of a set of definite clauses, whose heads all have the same predicate symbol; a goal is a set of negative literals. We consider a logic program to be a pair $\langle P, Q\rangle$ where P is a set of predicate definitions and Q is the *input*, which consists of a query, or goal, and possibly a set of facts for "database predicates" appearing in the program.

We follow the convention in deductive database literature of separating the set of rules with non-empty bodies (the set P) from the set of facts, or unit clauses, which appear in Q and are called the *database*. P is referred to as the *program*, or the set of rules. The motivation is that the rewriting algorithms to be discussed are applied only to the program, and not to the database. This is important in the database context since the set of facts can be very large. However, the distinction is artificial, and we may choose to consider (a subset of) facts to be rules if we wish. The meaning of a logic program is given by its least Herbrand model [103].

Following the syntax of Edinburgh Prolog, definite clauses (rules) are written as

$$p :- q_1, \ldots, q_n.$$

read declaratively as q_1 and q_2 and ...and q_n implies p. Names of variables begin with upper case letters, while names of non-variable (i.e., function and predicate) symbols begin with lower case letters. In addition, the following notation is used for lists: the empty list is written [ ], and a list with head H and tail L is written $[H|L]$. Note that control primitives such as Prolog's cut are not allowed.

3. The Bottom-Up Approach

The bottom-up approach that we consider consists of a two-part process. First, the program is rewritten in a form so that the bottom-up fixpoint evaluation of the program will be more efficient; next, the fixpoint of the rewritten program is computed by bottom-up iteration. Subsection 3.1 describes the initial rewriting, while Subsection 3.2 investigates the computation of the fixpoint of the rewritten program. Both these steps can be refined further; this is discussed in later sections.

3.1. The Magic Templates Rewriting Algorithm

As described in [13; 74], the initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or sips. For each rule, the associated sip determines the order in which the body literals are evaluated. In analogy to top-down evaluation, we can think of rules being used to generate answers to queries on the head predicate, with some arguments bound to given terms; different sips can be chosen for each head binding pattern.

We present a simplified version of the Magic Templates algorithm, tailored to the case that sips correspond to left-to-right evaluation with all arguments considered "bound" (perhaps to a free variable), as in Prolog. The idea is to compute a set of auxiliary predicates that contain the goals.

The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples.

DEFINITION 3.1. **The Magic Templates Algorithm**
Let P be a program, and $?-q(\overline{c})$ a query on the program. We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. Create a new predicate *magic_p* for each predicate p in P. The arity is that of p.
2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body.
3. For each rule r in P with head, say, $p(\bar{t})$, and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body contains the literal $magic\_p(\bar{t})$, and all literals that precede q_i in the rule.
4. Create a *seed* fact $magic\_q(\overline{c})$ from the query.

□

EXAMPLE 3.1. Consider the following program.

$$\begin{array}{lll} sg(X,Y) & :- & flat(X,Y). \\ sg(X,Y) & :- & up(X,U), sg(U,V), down(V,Y). \\ ?-sg(john,Z) & & \end{array}$$

For a choice of sips that orders body literals from left to right, as in Prolog, the Magic Templates algorithm rewrites it as follows:

$$\begin{array}{lll} sg(X,Y) & :- & magic\_sg(X,Y), flat(X,Y). \\ sg(X,Y) & :- & magic\_sg(X,Y), up(X,U), sg(U,V), \\ & & \qquad down(V,Y). \\ magic\_sg(U,V) & :- & magic\_sg(X,Y), up(X,U). \\ magic\_sg(john,Z). & & \end{array}$$

□

We present some results that characterize the transformed program P^{mg} with respect to the original program P, from [74]. The following theorem ensures soundness.

THEOREM 3.1. *[74] $\langle P,Q\rangle$ is equivalent to $\langle P^{mg},Q\rangle$ with respect to the set of answers to the query.*

DEFINITION 3.2. Let us define the *Magic Templates Evaluation Method* as follows:

1. Rewrite the program $\langle P,Q\rangle$ according to the choice of sips using the Magic Templates algorithm.

2. Evaluate the fixpoint of the rewritten program.

□

We hope that the slight abuse of notation in having the same name for the evaluation method and the rewriting algorithm will not lead to confusion; the distinction should be clear from the context. The second step above is presented in more detail in the next subsection. The rewriting has the important effect of mimicking Prolog in that (modulo optimizations such as tail recursion optimization and intelligent backtracking) only goals and facts generated by Prolog are generated. [2]

The careful reader will notice that some joins are repeated in the bodies of rules defining magic predicates and modified rules. The *supplementary* version of the rewriting algorithm essentially identifies these common subexpressions and stores them (with some optimizations that allow us to delete some columns from these intermediate, or supplementary, relations). We refer the reader to [13] for details.

Magic Templates is often presented along with an adornment rewriting that annotates predicates with a string composed of characters 'f' and 'b', with one character for each argument. This step, along with a modification of Magic Templates rewriting that projects out of magic predicates those arguments that have an f adornment, is used to ensure that the rewritten program generates only ground facts. For brevity, we omit this step.

3.2. Iterative Fixpoint Evaluation

We describe a refinement of bottom-up fixpoint evaluation called Semi-naive fixpoint evaluation, which is cited as the method of choice in the deductive database literature. Derivations made in one iteration are not repeated in subsequent iterations since in each iteration, only rule instantiations that utilize at least one new fact (i.e., one generated for the first time in the previous iteration) are considered.

We follow the presentation in [52] in the rest of this section, with some simplifications.

Let us first define a binary operator W_P, whose role is similar to that of the well-known T_P operator of [103]. We use square brackets ([...]) to denote multisets:

$$
\begin{aligned}
W_P(X,Y) = [h[\theta] \mid \; & h :- \; b_1, \ldots, b_k \text{ is a rule of } P, \\
& \theta \text{ is mgu of } (b_1, \ldots, b_k) \text{ and } (d_1, \ldots, d_k), \\
& Y \subseteq X, \{d_1, \ldots, d_k\} \subseteq X, \text{ and} \\
& k \geq 0 \text{ and } \{d_1, \ldots, d_k\} \cap Y \neq \emptyset.]
\end{aligned}
$$

[2] This is strictly true only for programs that do not generate non-ground facts. The method can be modified slightly to make the claim valid for all programs and to allow tail recursion optimization. For details, see [82].

The multiplicity of an element $h[\theta]$ is determined by the number of distinct rule instantiations of which it is the head literal. A more rigorous definition that formally accounts for the multiplicity is given in [52]; we have chosen to use an informal presentation for ease of exposition. Intuitively, W_P only allows deductions from the set of facts X that use the "new" facts Y, and "counts" a fact as often as it is derived (using distinct rule instantiations). We now define Semi-naive iteration. In the following definition, *set* is an operator that takes a multiset and returns a set, and *subs* is an operator that takes a multiset and returns an irredundant set. (An irredundant set, or *irrset*, is a set of elements such that no element subsumes another.)

DEFINITION 3.3. **Semi-naive Iteration (SN)**
Let $S_{-1} = S_0 = \delta_0 = \mathcal{F}$.

$$
\begin{aligned}
\delta_{n+1} &= dup\_elim(W_P(S_n, \delta_n - S_{n-1})) \\
S_{n+1} &= dup\_elim(S_n \cup \delta_{n+1}) \\
S &= \lim_{n\to\infty} S_n \\
GC &= set(S)
\end{aligned}
$$

where $\mathcal{F}$ is the set of facts, or rules with empty bodies, in the program P, and *dup_elim* is either *set* or *subs*. We refer to the variant with $dup\_elim = set$ as SN_S, and the variant with $dup\_elim = subs$ as SN_I. □

In Semi-naive iteration, the set of facts produced in iteration n (δ_n) is compared with the set of known facts (S_{n-1}) to identify the new facts produced ($\delta_n - S_{n-1}$). Duplicates generated within the same iteration are eliminated by the *set* operation. Only derivations that use one of these new facts are carried out in iteration $n+1$. This avoids generating many duplicate facts by avoiding repeated derivations. The algorithm terminates (at step $n+1$) when $S_{n+1} = S_n$; we must test whether $\delta_{n+1} \subseteq S_n$. Consequently, Semi-naive iteration terminates if and only if S is finite.

In this presentation we ignore the issue of how to ensure that only derivations that use a new fact are carried out. This is addressed in several papers (e.g. [6; 9; 5]). The problem becomes more complicated when it is desired to apply rules in a specified order in each iteration. This issue is discussed in [50; 79; 91].

The set GC is the set of *generated consequences* of the program.

Let *ground* be an operator that takes a set of possibly non-ground facts and a domain $\mathcal{D}$ and returns the set of ground facts containing only constants from $\mathcal{D}$ that are instances of the input set. The following result shows that the above iterative methods are consistent with the usual least Herbrand model semantics of [103]; here implicitly $\mathcal{D}$ is the domain for the program in question. We denote the least Herbrand model of the program by $\mathcal{M}$.

PROPOSITION 3.1. *[52] The set of generated consequences GC of a program computed using Semi-naive Iteration (either SN_S or SN_I) is such that $ground(GC) = \mathcal{M}$.* □

We note that ordinary fixpoint evaluation — frequently called "Naive" (N) iteration in the deductive database literature — corresponds to the case where the second argument of W_P in the definition of Semi-naive Iteration is replaced by S_n.

EXAMPLE 3.2. The following program illustrates the difference between N and SN.

$$
\begin{array}{lll}
d & :- & c. \\
c & :- & b. \\
b & :- & a. \\
a. & & \\
b. & &
\end{array}
$$

Note that SN_S and SN_I behave identically on this program. Both N and SN generate a and b in the first iteration. In the second iteration, SN generates c and a second occurrence of b, and eliminates the duplicate b. In the next iteration, it generates only d, since c was the only new fact found in the second iteration. In contrast, N additionally generates in each iteration all the facts that it generated in previous iterations. □

3.3. Related Work

The Alexander method was proposed independently of the Magic Sets approach in [84]. It is essentially the *supplementary* variant of the Magic Templates method, described in [13]. Seki has generalized the method to deal with non-ground facts and function symbols, and has called the generalized version Alexander Templates [92]. The Alexander methods always use a single left-to-right sip for each rule, for all possible goals.

The Magic Templates idea was developed in a series of papers ([7; 13; 74]). Several variants of the Magic Templates idea have also been proposed. For example, it is possible to compute supersets of the magic sets without compromising soundness. Although this results in some irrelevant computation, it may be possible to compute supersets more efficiently than the magic sets themselves [95].

Although the Magic Templates idea was introduced to deal with recursion, it provides significant improvements for non-recursive queries as well. In [57; 58] it is shown that the technique can be extended to deal with SQL programs, including those containing features like group-by, aggregation and arithmetic conditions. In [56] a performance evaluation, carried out on a DB2 relational system, is presented, demonstrating that the technique performs comparably to standard database techniques, and is often significantly better.

The Magic and Alexander methods are based on program transformations. Other methods use a combination of top-down and bottom-up control to propagate bindings. Pereira and Warren presented a memoing top-down evaluation procedure based on Earley deduction [69]. This evaluation procedure may be viewed as a top-down evaluation procedure that incorporates memoing. Vieille has proposed a method called QSQ [109; 110; 111] that can be viewed as follows. Goals are generated with a top-down invocation of rules, as in Prolog. However, there are two important differences: 1) whenever possible, goals and facts are propagated set-at-a-time, and 2) all generated goals and facts are memoed. If a newly generated goal is already memoed, this is recognized by duplicate elimination. Dietrich and Warren have proposed a method called Extension Tables [28]. This method is very similar to QSQ, but performs computation tuple at a time.

Finally, Kifer and Lozinskii have proposed a method called Filtering, which is based on constructing a rule-goal graph [47; 48]. There is a node in the graph for each predicate, and for each rule, and arcs from predicate nodes to each rule node in whose body it appears, and from rule nodes to the predicates that they define. The idea is to compute the fixpoint by propagating tuples along these arcs, and to restrict the computation by attaching "filters" to arcs.

The reader is referred to [64] for a more detailed discussion of related work.

4. Top-Down Versus Bottom-Up

In the past, bottom-up methods have not been seriously considered for the evaluation of logic programs because of a serious drawback: no techniques were known that avoided computing an unbounded number of irrelevant facts. Now that such techniques are known, it is worth reconsidering bottom-up approaches. They offer three significant advantages: (1) The declarative least Herbrand model semantics is guaranteed for positive Horn clause programs. (2) Redundant derivations are avoided through memoing, leading to asymptotic gains in efficiency for programs in which goals or facts can be derived in many ways. (3) As a consequence of (1), no operational guarantees need be made, thereby making possible a number of semantic optimizations. One example of a powerful optimization made possible by the declarative semantics is *factoring*, discussed in Section 5.

There are a number of top-down evaluation techniques for logic programs such as Prolog, the Query-Subquery (QSQ) approach and its extensions [109; 111], and Extension Tables [28]. SLD-AL resolution is a theoretical model of the execution of top-down evaluation techniques that perform memoization of facts. Memoing top-down evaluations are similar to the bottom-up approach with respect to advantages (1), (2), and to some extent (3) above. A natural question is how bottom-up evaluation techniques compare with memoing as well as non-memoing top-down evaluation techniques.

Initial comparisons were based on the number of facts derived by the different techniques. Thus, Ramakrishnan [74; 75] presents a class of evaluations and shows that within this class bottom-up evaluation of a program rewritten using Magic Templates computes an optimal number of facts. Seki [92] presents a direct comparison between the set of facts computed using Alexander Templates rewriting, and using SLD-AL resolution. Bry [16] shows that several top-down and bottom-up evaluation techniques can be viewed as specializations of a technique called Backward Fixpoint Procedure. Thus the least fixpoints of all these techniques have the same set of facts. Given a bottom-up evaluation that does not repeat derivation steps, these comparisons can be extended to include the number of derivations made.

There are three limitations to the above results. The first is that the comparisons do not include the cost of each inference. The second is that these comparisons ignore optimizations that are routinely performed on Prolog programs such as tail-recursion optimizations (Section 5.2). And third, the comparisons assume that all answers are required, and do not provide enough insight for the case that only one answer is needed (although there is no change in the worst case comparison).

Ullman [100] addressed the first problem, and compared the time cost of evaluation of Datalog programs using bottom-up evaluation, and a particular top-down evaluation technique. This comparison showed that for the class of safe Datalog programs (i.e., only ground facts are computed and no uninterpreted function symbols are allowed), bottom-up evaluation can be performed with a cost that is no more than that of the top-down evaluation in order of magnitude. Ramakrishnan and Sudarshan [81] compared a model of Prolog evaluation that performs tail recursion optimization with a model of bottom-up evaluation that uses a rewriting due to Ross [86] that performs tail recursion optimization. They showed that with this model of bottom-up evaluation the number of inferences and the time cost are (in order of magnitude) no more than that of Prolog, for a class of programs that includes safe Datalog and that allows ground terms to be constructed using uninterpreted function symbols, but with some restrictions. The rewritten program may generate non-ground facts, although the original program is not allowed to do so (for the class of programs considered). For the general case, where arbitrary non-ground terms are allowed, there are several problems that increase the cost per inference. These problems are faced not just by bottom-up evaluation schemes, but also by top-down evaluation schemes that perform memoization of facts. These issues are discussed in Section 5.10.

5. Some Important Program Optimizations

In this section, we survey a number of optimizations that apply to positive Horn programs and guarantee query equivalence in the least fixpoint model. That is, the same set of answers according to the declarative semantics of logic programs is computed. Note that no further guarantees are offered: no equivalence is guaranteed with respect to other program predicates, nor is any ordering preserved in the generation of answers.

This survey is not intended to be comprehensive, but rather to provide an indication of the flexibility that is available when we only need to preserve the declarative semantics. Ensuring that the declarative semantics is preserved has long been viewed as a bar to efficient evaluation; the results surveyed here indicate that it can sometimes be considerably more efficient to guarantee this semantics, rather than an operational semantics. For lack of space we do not cover some approaches to query optimization such as the semantic query optimization techniques proposed by Minker, King and others (see e.g. [49; 17]).

5.1. Predicate Factoring

The basic idea behind predicate factoring is to replace a predicate by two predicates of strictly smaller arity. This can result in significant speedups, as the example in this section illustrates. We present a simplified description that is tailored to the case of programs obtained by applying the Magic Templates algorithm described in Section 3.1. Our presentation is through the use of examples, and we do not describe sufficient conditions for the optimization to apply in general. We refer the reader to [65] for a detailed treatment.

In essence, we seek to take advantage of the magic predicates to replace the original predicate in P^{mg} by its projection onto its f argument positions. Thus, the original (non-magic) predicate in P^{mg} is factored into the magic predicate, which computes the b argument positions, and this new predicate, which computes the f argument positions.

EXAMPLE 5.1. We begin with a familiar example, transitive closure. While efficient algorithms are known, the rewriting algorithms presented in [66] were the first to automatically derive unary programs for single-selection queries, for all three forms (left-linear, right-linear, non-linear) of the recursive rule. We achieve the same result here by first applying the Magic Templates transformation and then factoring the rewritten program. To illustrate the technique, we consider a single program that includes all three forms of the recursive rule, although any one would suffice. It should be evident that we would also obtain a unary program if the original program contained just one of the recursive rules. Consider the program and single-selection query below:

$$
\begin{array}{lll}
t(X,Y) & :- & t(X,W), t(W,Y). \\
t(X,Y) & :- & e(X,W), t(W,Y). \\
t(X,Y) & :- & t(X,W), e(W,Y). \\
t(X,Y) & :- & e(X,Y). \\
query(Y) & :- & t(5,Y).
\end{array}
$$

The Magic Templates algorithm rewrites this to:

$$
\begin{array}{lll}
magic\_t^{bf}(W) & :- & magic\_t^{bf}(X), t^{bf}(X,W). \\
magic\_t^{bf}(W) & :- & magic\_t^{bf}(X), e(X,W). \\
magic\_t^{bf}(5). & & \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), t^{bf}(X,W), t^{bf}(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), e(X,W), t^{bf}(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), t^{bf}(X,W), e(W,Y). \\
t^{bf}(X,Y) & :- & magic\_t^{bf}(X), e(X,Y). \\
query(Y) & :- & t^{bf}(5,Y).
\end{array}
$$

If we identify $magic\_t^{bf}$ tuples with goals in a top-down evaluation, we see that only the last occurrence of t^{bf} in a rule body generates new goals, and further, the answer to a new goal is also an answer to the goal that invoked the rule. In fact, every answer to a subgoal is also an answer to the query goal $magic\_t^{bf}$. A second observation is that if c is generated as an answer to a subgoal, then a new subgoal $magic\_t^{bf}(c)$ is also generated. These observations lead us to conclude that it does not matter to which subgoal an answer corresponds; its role in the computation is the same in any case. That is, $t^{bf}(X,Y)$ can be factored into $bt(X)$ and $ft(Y)$ in the Magic program. Doing this and applying some simple syntactic optimizations, which are discussed in [65], we finally obtain the following unary program:

$$
\begin{array}{lll}
magic\_t^{bf}(W) & :- & ft(W). \\
magic\_t^{bf}(5). & & \\
ft(Y) & :- & magic\_t^{bf}(X), e(X,Y). \\
query(Y) & :- & ft(Y).
\end{array}
$$

□

Predicate factoring, as described above, requires the entire program to satisfy the required conditions. Kemp et al. [42] generalized this transformation so that it can be applied even if a subprogram satisfies the required conditions; this allows for sharing of subcomputation across multiple calls to a factored predicate.

5.2. Tail Recursion Optimization

An optimization that is closely related to factoring is tail-recursion optimization. Consider a program of the form

$$p(0,1).$$
$$p(X,Y) :- X > 0, p(X-1,Y).$$

If there is a query $?-p(10,X)$, queries $?-p(9,X)$ and so on till $?-p(0,X)$ are set up. At this point an answer $p(0,1)$ is generated by bottom-up evaluation of a program rewritten using Magic Templates. However a Prolog system can recognize that the second rule is "tail-recursive," and can avoid generating answers for intermediate goals. The reason is that any answer substitution for the last goal results in an answer for the head of this rule with the same answer substitution. Applying this optimization for each goal on this rule, when the goal $?-p(0,X)$ succeeds, an answer $p(10,1)$ is created directly, rather than an answer $p(0,1)$.

Ross [86] shows how the same optimization can be performed in bottom-up evaluation. The details are beyond the scope of this paper, but the essential idea is to generate goal facts of the form $query(p1(\overline{t1}), p2(\overline{t2}))$; such a fact says the first argument is a subgoal for $p1$, and any answer substitution that solves it should be used to generate answers for $p2(\overline{t2})$ rather than for $p1(\overline{t1})$.

5.3. Projecting Arguments

Query optimization for relational database queries exploits the commutativity of *selection* and *projection* operators with respect to the *join* operator whenever possible, in order to reduce the size of relations that are being joined. This is often referred to as "pushing" selections and projections. Pushing selections is achieved through the use of the Magic Templates transformation; the introduction of recursion makes it necessary to compute auxiliary sets. Pushing projections has also been explored, and the gains can be significant. We will illustrate the idea through examples, and refer the reader to [76] for details.

EXAMPLE 5.2. Consider the transitive closure program of Example 1.1, but with the query $query(Y) :- t(\_, Y)$. The underscore "_" indicates that we do not care about the value in the first argument position; we simply want the set of values that appear in the second argument position of t (with some arbitrary value in the first argument position). Note that such queries are likely to arise during the course of query optimization. □

An adornment that distinguishes don't-care argument positions (d) from the rest (needed or n) was used in [76] to push projections.

EXAMPLE 5.3. Consider a simplified version of the transitive closure program:

$$\begin{array}{lll} t(X,Y) & :- & t(X,W), e(W,Y). \\ t(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t(\_,Y). \end{array}$$

The adorned program — using n and d adornments — is:

$$\begin{array}{lll} t^{dn}(X,Y) & :- & t^{dn}(X,W), e(W,Y). \\ t^{dn}(X,Y) & :- & e(X,Y). \\ query(Y) & :- & t^{dn}(\_,Y). \end{array}$$

It is easy to see that the d argument positions can be uniformly deleted, and this leaves us with a program in which the recursive predicate is unary. □

The previous example illustrated how the adornment algorithm can sometimes push the projection through recursion and thereby reduce the arity of recursive predicates. (The observation that pushing projections could reduce arity of recursive predicates was first made by Aho and Ullman [1], and later Kifer and Lozinskii [47] and Apers et al. [2]. The adornment algorithm in [76] generalizes their results.) The acute reader will have observed that more can be achieved — the recursive rule may be deleted entirely. Algorithms for deleting redundant rules and literals can be utilized to detect this; in fact, since such opportunities are frequently created by pushing projections, we can devise special algorithms for rule and literal deletion that exploit this. The following program is obtained in the above example using the techniques of [76]:

$$\begin{array}{lll} t^{dn}(Y) & :- & e(X,Y). \\ query(Y) & :- & t^{dn}(Y). \end{array}$$

5.4. Counting

Counting ([7; 88; 13]) is a refinement of the Magic Sets approach. Whereas the Magic Sets method restricted the computation to relevant facts, Counting additionally computes indices for each fact that indicate why it is relevant, and this additional information is used to delete some literals from rule bodies and to reduce the arity of some predicates by deleting some argument positions.

Goals correspond to magic facts, as we noted earlier. In the Counting program, magic facts are called *count* facts. The index value for a non-count fact is simply the index value of the goal for which it was generated as an answer. Index values can be encoded in various ways; we will not consider the details.

EXAMPLE 5.4. Consider the program of Example 3.1 again. The Counting algorithm generates:

$$
\begin{array}{ll}
sg^{bf}(Y, I) & :- \ count\_sg^{bf}(X, I), flat(X, Y). \\
sg^{bf}(Y, I) & :- \ sg^{bf}(V, I+1), down(V, Y). \\
count\_sg^{bf}(U, I+1) & :- \ count\_sg^{bf}(X, I), up(X, U). \\
count\_sg^{bf}(john, 0). &
\end{array}
$$

We essentially compute (i) "count" facts (analogous to "magic" facts) that are numbered with their distance from the query (the *count* facts), and (ii) values in the free argument positions of program (*sg*) facts (which we will refer to as "answers" in this example), also numbered according to the "count" fact used to generate them. Thus, we only compute sets of goals and answers, all numbered according to their distance from the query. The answers to the original query are the (new, reduced arity) *sg* facts at distance (index value) 0. The key observation here is that we can no longer identify the set of answers to any intermediate subgoals since we only maintain sets of goals and sets of facts, and there may be several goals at a given distance (except 0, where the only goal is the original query). The potential gain in the method derives from this observation — where several goals have the same, or largely overlapping, answer sets we gain by not associating an answer with each of these goals. On the other hand, it is possible to lose in performance if each goal/answer appears at several distances from the query. This can happen for example if there is a cycle in the *up* relation. Indeed, such a cycle can lead to non-termination. It is usually suggested that if a goal is derived with two different distances (i.e. two *count* facts differ only in the index values), the Counting method should be abandoned in favor of Magic Sets. □

5.5. Space Optimization

Bottom-up evaluation derives new facts, but has no mechanism built in to discard facts during the evaluation — all facts are retained till the end of the evaluation. It is important to discard facts during an evaluation, once they are not needed, for otherwise space requirements may grow in an unbounded fashion. Facts are needed to make derivations and to detect duplicate derivations of facts (which may be needed for termination). Consider the following program which computes fibonacci numbers:

$$
\begin{array}{l}
R1 : fib(0, 1). \\
R2 : fib(1, 1). \\
R3 : fib(N, X1 + X2) :- \ N > 1, fib(N-1, X1), fib(N-2, X2).
\end{array}
$$

In a Semi-naive evaluation of this program, once a fact $fib(n, \_)$ is deduced, there is no need to keep any facts $fib(m, \_)$ for $m \leq n-2$, since all derivations using such facts have been made, and the facts will not be derived again. Such facts can then be discarded during the evaluation. This issue is studied in detail in [63; 98], where several sufficient conditions for discarding facts are developed.

5.6. Duplicate Elimination

In general, when a fact is derived in a bottom-up evaluation, duplicate checking needs to be done to see if the fact was derived earlier. If it was not, derivations can be made using the fact. If it was, the fact can be discarded. This check is important for termination of many programs. However, for some programs such as the fibonacci program above, with Semi-naive bottom-up evaluation, duplicate facts are never generated. Hence duplicate elimination need not be done for such programs. Maher and Ramakrishnan study this issue in detail in [52], and provide sufficient conditions to detect if a program will not compute duplicate facts.

5.7. Linearizing Programs

An interesting class of program transformations has recently been explored by a number of researchers [39; 114; 89; 78]. The objective is to transform a program that contains non-linear rules into an equivalent one that contains only linear rules; this may make some of the other transformations surveyed in this paper applicable, or permit simplifications in the implementation of the fixpoint evaluation phase. We do not consider these results here due to space limitations.

5.8. Bounded Recursion and Redundant Literals

One interesting optimization that has been studied in some detail is that of *bounded recursion*, where a recursive program is bounded if it can be replaced by a nonrecursive program.

Note that here we are not asking when recursion can be replaced by iteration; rather, we are asking when a logic program containing recursive clauses has an equivalent finite logic program in which no clause is recursive.

EXAMPLE 5.5. The following example is from [62].

$$buys(X,Y) :- likes(X,Y).$$
$$buys(X,Y) :- trendy(X), buys(Z,Y).$$

In English, a person X buys a product Y if either X likes Y, or X is trendy and a person Z has bought Y. This recursive program is equivalent to the following nonrecursive program:

$$buys(X,Y) :- likes(X,Y).$$
$$buys(X,Y) :- trendy(X), likes(Z,Y).$$

□

In general, detecting bounded recursions is undecidable. A rich classification of sets of programs for which detecting boundedness is decidable or undecidable has been developed in the literature; see, for example, [26; 38; 31; 62; 67; 106]. A related problem is to identify literals and rules that are redundant, i.e., that do not change the set of answers for any input database. For more details about detecting and eliminating redundancy from recursive Datalog programs, see [68].

5.9. Algebraic Properties of Programs

The fixpoint evaluation of a logic program can be refined by taking certain algebraic properties of the program into consideration. Such refinements, and techniques for detecting when they are applicable, have been investigated by several researchers [37; 39; 51; 61; 78]. Ioannidis presents an algebraic formulation of Datalog programs that is particularly suited to reasoning about such properties of programs [39]. The following idea is illustrative of this class of optimizations.

EXAMPLE 5.6. We begin with an example that illustrates *commutativity* of rules. The idea has been studied in the references cited above, and several sufficient conditions are known for detecting when this property holds.

$$\begin{aligned} r_1 : p(X,Y) \; &:- \; a(X,Z), p(Z,Y). \\ r_2 : p(X,Y) \; &:- \; p(X,Z), b(Z,Y). \\ r_3 : p(X,Y) \; &:- \; c(X,Y). \end{aligned}$$

It can be shown that applications of rules r_1 and r_2 commute. That is, $r_1.r_2 = r_2.r_1$. This fact can be utilized to avoid many redundant derivations by evaluating the fixpoint of the program as follows: first apply rule r_3, then apply r_2 as often as possible, and finally apply r_1 as often as possible. □

5.10. Non-Ground Facts

Non-ground facts are useful for several reasons. They are often useful in knowledge representation, where we can say that a fact is true for all values of a variable. They are useful in a database that stores (and possibly manipulates) rules. They are also useful for specifying patterns in queries, and are necessary for representing constraint facts.

However, supporting non-ground facts in a bottom-up evaluation (actually, in general in an evaluation that memos facts) has an associated cost. (1) When making an inference, facts have to have their variables renamed so that they do not share variable names. (2) When a head fact with a large term as an argument is generated, we must share the structure of subterms with facts used to derive it, or do extensive copying. This is easy with ground terms, but with non-ground terms structure sharing becomes more difficult. (3) If variables in a large term get instantiated during the inference, creation of the head fact is made more complicated. (4) Unification of large non-ground terms using currently known techniques is inefficient, especially compared to the cost of unification for ground terms using hash-consing [36; 90]. (5) Indexing of facts containing variables using currently known techniques is inefficient, especially compared to the cost of indexing ground facts using, say, hash tables.

Consider a program to append two lists:

$$
\begin{array}{l}
append([], Z, Z). \\
append([X|Y], Z, [X|Y1]) :- append(Y, Z, Y1). \\
\text{Query: ?-}append([A, B, C], [D, E, F], ANS).
\end{array}
$$

The Magic Templates rewriting of the above program is

$$
\begin{array}{ll}
append([], Z, Z) & :- magic\_append([], Z). \\
append([X|Y], Z, [X|Y1]) & :- magic\_append([X|Y], Z), \\
 & \qquad append(Y, Z, Y1) \\
magic\_append([A, B, C], [D, E, F]). &
\end{array}
$$

If the query uses ground lists, this program runs in $O(n)$ time, where n is the size of the lists to be appended. If the query uses non-ground lists, and the program is evaluated without any special optimizations for non-ground terms, the evaluation takes $O(n^2)$ time.

The structure sharing scheme of Boyer and Moore [14] addresses the first three problems. The idea is to maintain binding environments with terms, and note variable bindings there; this also makes sharing subterms easier. A somewhat complicated naming scheme helps rename all variables in a term in constant time. However, in the worst case, the cost of looking up variable bindings is large since many binding environments may need to be searched. Also, this scheme does nothing to solve the fourth problem, namely the cost of unification.

There appears to be no general solution for theorem proving systems, which is the context in which these problems first appeared. However, in special cases of bottom-up evaluation of logic programs, efficient techniques are possible.

In the case of the append program there is no need to do renaming for the first and last rule of the magic program. For the second rule if the terms to which the variables Y and Z are bound by the first literal are exactly the same (pointer match), there is no need for renaming - the variables in the term to which $Y1$ is bound can be deduced to be the same as those in the terms that Y and Z are bound to. This same optimization can perform unification in constant time. Efficient indexing techniques are possible when

Supplementary Magic Templates rewriting is used. No variable instantiation occurs within large terms, and this makes sharing of subterms easy. Putting these optimizations together, the program can be evaluated in $O(n)$ time. A class of programs for which these optimizations are possible is discussed in Sudarshan and Ramakrishnan [97].

6. Extensions to the Horn Clause Logic Programming Paradigm

6.1. Negation

A deductive database query language can be enhanced by permitting negated literals in the bodies of rules in programs. This provides the ability to deal with non-monotonic deduction. However, in the presence of negated literals, a program may not have a least model. For example, the program:

$$p(a) :- \neg p(b).$$

has two minimal models, $\{p(a)\}$ and $\{p(b)\}$, but no least model. The meaning of a program with negation is usually given by some "intended" model ([18; 3; 72; 71; 34; 85; 73; 104], among others).

One important class of negation that has been extensively studied is stratified negation [18; 3]. Intuitively, a program is stratified if there is no recursion through negation. Programs in this class have an intuitive semantics and can also be efficiently evaluated. The following example describes a stratified program.

EXAMPLE 6.1. Consider the following program $P2$:

$$\begin{aligned}
r1 : anc(X,Y) &:- par(X,Y).\\
r2 : anc(X,Y) &:- par(X,Z), anc(Z,Y).\\
r3 : nocyc(X,Y) &:- anc(X,Y), \neg anc(Y,X).
\end{aligned}$$

Intuitively, this program is stratified because the definition of the predicate *nocyc* depends (negatively) on the definition of *anc*, but the definition of *anc* does not depend on the definition of *nocyc*.

A bottom-up evaluation of $P2$ would proceed SCC-by-SCC. First, we would compute a fixpoint of rules $r1$ and $r2$ (the rules defining *anc*). Rule $r3$ is applied only when the all *anc* facts are known. This can be seen to be as efficient as the evaluation of programs without negation.

However, the Magic Templates transformation of a stratified program may become unstratified, and the evaluation of the resulting unstratified program has received considerable attention in the literature ([4; 45; 43; 10], among others). □

A natural extension of stratified programs is the class of locally stratified programs [72]. Intuitively, a program P is locally stratified if the (propositional) program $Gr(P)$ obtained by taking all ground instances of all rules in P is stratified. Local stratification has been extended to modular stratification in [85]. A program P is said to be modularly stratified if each strongly connected component (SCC) of P is locally stratified after removing instantiated rules containing literals that are false in lower SCCs.

EXAMPLE 6.2. Consider the following program $P3$:

$$r1 : even(0).$$
$$r2 : even(s(X)) :- \neg even(X).$$

This program can be seen to be locally stratified. Consider the following variant $P4$ of the above program:

$$r1 : even(0).$$
$$r2 : even(X) :- succ(X, Y), \neg even(Y).$$
$$succ(1, 0). \quad succ(2, 1). \quad succ(3, 2).$$

Since rule $r2$ can be instantiated to give the same value for X and Y, program $P4$ is not locally stratified. However, it is modularly stratified. Locally and modularly stratified programs can be evaluated using the techniques described in [85; 43]. The evaluation of the Magic Templates transformation of this class of programs has also been considered in the literature ([85; 43; 80]). □

6.2. Set Grouping and Aggregation

The *group-by* construct is used in SQL to partition a relation based on the values of some of the attributes of the relation. A variant of this construct is used in LDL [11; 60] to create nested sets within facts. We use the syntax of LDL in our discussion; however, the semantics we use is from the CORAL[77] system, and is a minor variant of the LDL semantics. The following example illustrates the use of set grouping construct $< >$:

$$multiset\_of\_grades(Class, < Grade >) :- student(Name, Class, Grade).$$

We assume that there is at most one occurrence of the $< >$ operator in the head of any rule, and the operator does not occur in the body. We also assume for now that the program is stratified wrt the $< >$ operator, i.e., the definition of the predicate in the body does not depend on the head predicate if the head of the rule has an occurrence of the $< >$ operator. We first (conceptually) create a multiset of tuples for $multiset\_of\_grades$ using the rule

$$multiset\_of\_grades(Class, Grade) :- student(Name, Class, Grade).$$

Now for each value of $Class$ (in general, each value of the arguments of the head that are not enclosed in the $< >$), we create a multiset containing all the corresponding values for $Grade$. For each value of $Class$ let this multiset be called S_{Class}; we then create a fact $multiset\_of\_grades(Class, S_{Class})$ for each value of $Class$.

Aggregate operations such as $count$, sum, min, and max, which can be applied to sets of values, are permitted in relational database query languages such as SQL. These can be incorporated into logic programming too, using the set grouping construct to create multisets that are aggregated on. Extending the above example, consider the following rule: The following example illustrates the use of aggregation:

$$max\_grade(Class, max < Grade >) :- student(Name, Class, Grade)$$

As before, for each value of $Class$ we create a multiset. But now we apply the aggregate operation max to the multiset, and create a head fact using this value rather than the set itself.

There are several approaches to relaxing the assumption of stratification. Several of the approaches used for negation carry over to set grouping. There are approaches based on weaker forms of stratification such as group stratification and magical stratification [57], or modular stratification [85]. Extensions of the well-founded and stable models to deal with aggregates are considered in [44]. Monotonic programs, where a derivation using an incomplete set does not affect the set of facts computed, are discussed in [25; 57].

In certain contexts, an incomplete set may be used to generate facts that would not be generated using a complete set, without affecting the final answer to some queries on a program. One such context is when subsumption of facts is defined in such a way that facts created using incomplete sets will be found to be subsumed and will be discarded. Ross and Sagiv [87] examine such classes of programs. Cruz and Norvell [27] examine the use of such forms of aggregation in the context of transitive closure. Ganguly, Greco and Zaniolo [33] examine how to evaluate a class of programs that are monotonic in a sense different from that used in [57]. Ganguly et al. [33] and Sudarshan and Ramakrishnan [96] examine rewrite optimizations of programs that use aggregation, and the rewriting can result in unstratified programs. The following program for computing the shortest path length benefits from these optimizations:

$$\begin{array}{ll}
R1 : s_p_length(X, Y, min(< C >)) & :- \ path(X, Y, C). \\
R2 : path(X, Y, C1) & :- \ path(X, Z, C), \\
 & \quad edge(Z, Y, EC), \\
 & \quad C1 = C + EC. \\
R3 : path(X, Y, C) & :- \ edge(X, Y, C).
\end{array}$$

Query: ?-$s\_p\_length(X, Y, C)$.

For the above program, both the above techniques reduce the number of facts computed from possibly infinite to polynomial, and both evaluation techniques can be implemented to be as efficient as Dijkstra's algorithm for shortest paths.

6.3. Constraint Query Languages

Constraint query languages ([41; 83; 23; 8] among others) differ from deductive database query languages in several ways:

1. With each rule r in a constraint query language, we can have a conjunction of constraints over some algebraic structure, in addition to the other literals in the body of the rule.
2. Each fact $p(\overline{X}; C)$ is a conjunction of constraints over the given structure, a relation is a finite collection of such facts (a disjunction of conjunctions of constraints), and a database is a finite collection of such relations.

Constraint query languages provide a model for reasoning about (potentially) infinite sets of ground facts using finite representations. This is because a constraint fact $p(\overline{X}; C)$ is a finite representation of the (potentially) infinite set of ground facts that satisfy the conjunction of constraints C.

Jaffar and Lassez [40] gave a model theory, a fixpoint theory, and a top-down operational semantics for such programs. Kanellakis et al. [41] gave a bottom-up operational semantics for some classes of constraint query languages.

EXAMPLE 6.3. Consider the following constraint logic program P with a query:

$$\begin{array}{l}
r1 : p(X) \ :- \ X \leq 5, p1(X), p2(X). \\
r2 : p(X) \ :- \ X \geq 7, p2(X). \\
r3 : p1(X) :- \ X \geq 1, X \leq 6. \\
r4 : p2(X) :- \ X \geq 3, X \leq 9. \\
r5 : p2(X) :- \ X \geq 11, X \leq 15.
\end{array}$$

Query: ?-$p(X), C$.

where p is the query predicate, and C is a conjunction of constraints on the query. The least model of the program P is the (infinite) set of ground facts $\{p1(a), 1 \leq a \leq 6; p2(b), 3 \leq b \leq 9; p2(c), 11 \leq c \leq 15; p(d), 3 \leq d \leq 5; p(e), 7 \leq e \leq 9; p(f), 11 \leq f \leq 15\}$.

Given the above query, program P can be rewritten using Magic Templates and left-to-right sips for each rule to obtain $\langle P^{mg}, Q^{mg}\rangle$ as follows:

$$
\begin{array}{lll}
r1: & p(X) & :- \ magic\_p(X), X \leq 5, p1(X), p2(X). \\
r2: & p(X) & :- \ magic\_p(X), X \geq 7, p2(X). \\
r3: & p1(X) & :- \ magic\_p1(X), X \geq 1, X \leq 6. \\
r4: & p2(X) & :- \ magic\_p2(X), X \geq 3, X \leq 9. \\
r5: & p2(X) & :- \ magic\_p2(X), X \geq 11, X \leq 15. \\
r6: & magic\_p1(X) & :- \ magic\_p(X), X \leq 5. \\
r7: & magic\_p2(X) & :- \ magic\_p(X), X \leq 5, p1(X). \\
r8: & magic\_p2(X) & :- \ magic\_p(X), X \geq 7. \\
Q^{mg}: & magic\_p(X) & :- \ C.
\end{array}
$$

We associate the desired query constraints with the corresponding seed magic fact.

The magic program is also a constraint logic program. The magic predicates can be considered to define a relation consisting of the subgoals that a top-down evaluation ([40]) would generate, and the constraints associated with each magic fact correspond to the desired constraints for answers to that subgoal.

We describe the evaluation of the magic program $\langle P^{mg}, Q^{mg}\rangle$ for various constraints C:

- $C \equiv (X \geq 7) \& (X \leq 12)$:
  All $p(a)$ facts, $7 \leq a \leq 9$, and $p(b)$ facts, $11 \leq b \leq 12$ are in the least model of P, and are answers to the query.
  In the bottom-up evaluation of the magic program, $magic\_p(X; X \geq 7, X \leq 12)$ is first derived (using rule Q^{mg}). Rule $r8$ can now be applied to derive $magic\_p2(X; X \geq 7, X \leq 12)$. (Rule $r6$ cannot be applied because the constraints in the body of the rule are not satisfied.) This fact can now be used in rules $r4$ and $r5$ to derive $p2(X; X \geq 7, X \leq 9)$ and $p2(X; X \geq 11, X \leq 12)$. These $p2$ facts can be used in the body of rule $r2$ to derive $p(X; X \geq 7, X \leq 9)$ and $p(X; X \geq 11, X \leq 12)$.
  This gives a finite representation of the desired answers to the query, and mimics the top-down evaluation as well. Note that the bottom-up evaluation does not compute any facts for $p1$ since these are irrelevant to answering the query.
- $C \equiv X \leq 2$:
  There are no p facts in the least model that are answers to the query. In the bottom-up evaluation of the magic program, $magic\_p(X; X \leq 2)$ is first derived. Rule $r6$ can now be applied to derive $magic\_p1(X; X \leq 2)$. (Rule $r8$ is not applicable.) This fact can be used in the body of rule $r3$

to derive $p1(X; X \geq 1, X \leq 2)$. Rule $r7$ can now be applied to derive $magic\_p2(X; X \geq 1, X \leq 2)$. However, this fact cannot be used in either rule $r4$ or rule $r5$ and the evaluation terminates.
Again, this mimics the top-down evaluation. □

6.4. Quantitative Logic Programs

Quantitative logic programs (also referred to as fuzzy databases) ([94; 102; 46], among others) differ from ordinary logic programs in several ways:

1. With each fact $p(\overline{a})$, we associate a "certainty," $0 \leq \alpha \leq 1$, which is a measure of the strength of the evidence for $p(\overline{a})$.
2. With each rule r in a quantitative logic program, we associate a certainty function f_r that is used to compute the certainty of the head fact, given the certainties of facts in the body.
3. With each predicate p, we associate an evidence combination function F_p that combines the strengths of evidences obtained from different sources all supporting the same fact.

This provides a model for rule-based reasoning in expert systems, where the usual logical reasoning in terms of the truth values **true** and **false** is insufficient.

We describe the restricted case of K-standard sequence logics ([94; 102]), and refer the interested reader to Kifer and Li [46] for more general types of quantitative logic programs. Quantitative logic programs based on K-standard sequence logics were considered by Van Emden [102], who gave a model theory, a fixpoint theory, and a top-down operational semantics for such programs. A rule in a quantitative logic program with K-standard sequence logic is of the form:

$$r : p(\overline{X}) : c_r * \min\{\alpha_1, \alpha_2, \ldots, \alpha_n\} :- p1(\overline{X_1}) : \alpha_1, \ldots, p_n(\overline{X_n}) : \alpha_n.$$

Thus, the rule certainty function, f_r, associated with rule r is

$$f_r = c_r * \min\{\alpha_1, \ldots, \alpha_n\}$$

where c_r is some constant, $0 < c_r \leq 1$, associated with the rule. The evidence combination function $F_p(\{\alpha_1, \ldots, \alpha_m\})$ in such a logic is defined as $\max\{\alpha_1, \ldots, \alpha_m\}$.

EXAMPLE 6.4. Consider the following program $P1$ in K-standard sequence logic, with query Q:

$r1 : p(X) : 0.5 * \min\{\alpha_1, \alpha_2\} :- p1(X) : \alpha_1, p2(X) : \alpha_2.$
$r2 : p(X) : 0.45 * \alpha_1 \quad :- p2(X) : \alpha_1.$
$r3 : p1(1) : 0.4.$
$r4 : p2(1) : 0.5.$
$r5 : p2(2) : 0.25.$
Query: $?\text{-}p(1) : c_p.$

where p is the query predicate, and c_p is the desired certainty for $p(1)$. The least model of the program $P1$ is $\{p1(1) : 0.4, p2(1) : 0.5, p2(2) : 0.25, p(1) : 0.225, p(2) : 0.1125\}$.

Given the above query, program $P1$ can be rewritten using (a variant of) Magic Templates and left-to-right sips for each rule to obtain $\langle P1^{mg}, Q^{mg}\rangle$ as follows:

$$
\begin{array}{lll}
r1: & p(X) : 0.5 * \min\{\alpha_1, \alpha_2\} & :- \ magic\_p(X) : \alpha_0, p1(X) : \alpha_1, \\
& & \quad p2(X) : \alpha_2, \\
& & \quad \alpha_0 \leq 0.5 * \min\{\alpha_1, \alpha_2\}. \\
r2: & p(X) : 0.45 * \alpha_1 & :- \ magic\_p(X) : \alpha_0, p2(X) : \alpha_1, \\
& & \quad \alpha_0 \leq 0.45 * \alpha_1. \\
r3: & p1(1) : 0.4 & :- \ magic\_p1(1) : \alpha_0, \alpha_0 \leq 0.4. \\
r4: & p2(1) : 0.5 & :- \ magic\_p2(1) : \alpha_0, \alpha_0 \leq 0.5. \\
r5: & p2(2) : 0.25 & :- \ magic\_p2(2) : \alpha_0, \alpha_0 \leq 0.25. \\
r6: & magic\_p1(X) : \alpha_0/0.5 & :- \ magic\_p(X) : \alpha_0. \\
r7: & magic\_p2(X) : \alpha_0/0.5 & :- \ magic\_p(X) : \alpha_0, p1(X) : \alpha_1, \\
& & \quad \alpha_1 \geq \alpha_0/0.5 \\
r8: & magic\_p2(X) : \alpha_0/0.45 & :- \ magic\_p(X) : \alpha_0. \\
\multicolumn{3}{l}{Q^{mg} : magic\_p(1) : c_p.}
\end{array}
$$

We associate the desired query certainty with the corresponding seed magic fact $magic\_p(1)$.

Note that the magic program cannot be viewed as a quantitative logic program in K-standard sequence logic. The magic predicates can be considered to define a relation consisting of the subgoals that a top-down evaluation would generate, and the number associated with the magic fact corresponds to the certainty desired for answers to that subgoal—this number can thus be semantically viewed as a "cutoff."

We describe the evaluation of the magic program $\langle P1^{mg}, Q^{mg}\rangle$ for various values of c_p:

- $c_p = 0.2$:
  Since $p(1) : 0.225$ is in the least model of $P1$, so is $p(1) : 0.2$. The desired answer to the query is **yes**.
  In the bottom-up evaluation of the magic program, $magic\_p(1)$ is first derived (using rule Q^{mg}) with a cutoff of 0.2. Rule $r6$ can now be applied to derive $magic\_p1(1)$ with a cutoff of 0.4, and rule $r8$ can be applied to derive $magic\_p2(1)$ with a cutoff of 0.444. Rule $r3$ can then be applied to derive $p1(1)$ with certainty 0.4 (since the constraint in the body is satisfied); rule $r4$ can also be applied to derive $p2(1)$ with certainty 0.5. Now, the magic fact $magic\_p2(1)$ also can be derived using rule $r7$ with a cutoff of 0.4. The fact $p2(1) : 0.5$ is now derived again. Finally, rule $r1$ can be applied to compute $p(1) : 0.2$, and rule $r2$ can be applied to compute $p(1) : 0.225$.

This gives the required answer to the query, and mimics the top-down evaluation ([102]) as well. Note that the bottom-up evaluation does not compute facts $p2(2) : 0.25$ and $p(2) : 0.1125$ which are irrelevant to answering the query.

– $c_p = 0.25$:
The desired answer to the query is **no**. In the bottom-up evaluation of the magic program, $magic\_p(1)$ is first derived with a cutoff of 0.25. The magic fact $magic\_p1(1)$ can now be derived (using rule $r6$) with a cutoff of 0.5 (indicating that we need to compute $p1$ with a certainty of at least 0.5); $magic\_p2(1)$ can also be derived (using rule $r8$) with a cutoff of 0.556. However, neither $p1(1)$ nor $p2(1)$ can be derived, since the constraints in the appropriate bodies (of rules $r3$ and $r4$) fail. Since no $p1$ or $p2$ fact is derived, no further derivations are made, and there is no answer to the query (as expected).
Again, this mimics the top-down evaluation, and no irrelevant facts are computed. □

7. Deductive Database System Implementations

There have been a number of implementations of deductive databases. In this section we briefly discuss some of them.

One of the early implementations was LDL [60; 20; 21]. LDL is a main-memory database system implemented at MCC Austin. It extends the declarative semantics of Horn clauses by allowing set grouping and aggregate operations. The treatment of sets allows set-terms to be specified in the body of rules; set-matching is used to support this feature. It also allows rules to perform updates, which means that such rules do not have the usual declarative semantics. Rather, their semantics is defined in a constructive fashion [59]. LDL also provides some control primitives such as choice, provides a module feature for organizing code, and provides primitives for integration with C.

NAIL! [99; 54; 53] was another early deductive database system. The current version is compiled into a set-oriented procedural language called Glue [70]. NAIL! provides higher order syntactic features, using the semantics of Hilog [19]. Glue/NAIL! is currently implemented on top of Prolog, and is main-memory oriented.

EKS-V1 [108] is a deductive database system that combines features of disk-oriented and in-memory databases. It is implemented on top of the MEGALOG logic programming system which supports access to secondary storage. It provides facilities for handling negation and aggregation in a set-oriented fashion. Unlike the other deductive database systems discussed here, it provides support for integrity constraint maintenance and hypothetical reasoning.

Aditi [101] is a deductive database system that is primarily disk oriented. It provides support for parallelization of the low level operations such as joins. It is integrated with Prolog, and is compiled into an intermediate language which is then interpreted. All relational operations are performed with relations assumed to be disk resident, and join techniques such as sort-merge and hash-join are used, which means that Aditi is aimed at applications that are I/O intensive, but typically perform few iterations.

Starburst SQL (see e.g. [57]) extends SQL with recursive view definitions. This provides greater expressive power to SQL users, while providing the I/O optimization facilities of SQL in a direct manner for recursive applications.

CORAL [77] is a deductive database system developed by the authors, and is described in detail in a separate section. LDL, NAIL! and CORAL belong in one family of deductive database systems in that they are main-memory oriented. They attempt to provide the power of logic programming using complex terms built using uninterpreted function symbols, and add new constructs to standard Horn clause logic programs. EKS-V1, Aditi and Starburst SQL on the other hand are oriented more towards secondary storage, and provide less support for complex terms.

8. CORAL—A Deductive Database Programming Language

CORAL[3] is a database programming language being developed at UW-Madison. An important goal of CORAL is to provide a powerful declarative query language. A second goal of CORAL is to provide a clean interface between an imperative language (C++) and the declarative language. This is necessary for several reasons such as permitting the user to perform actions such as updates and to implement time critical functions more efficiently. To this end C++ is extended with several types, constructs and library functions.

A CORAL program can be viewed as a collection of *declarative modules*, and an *imperative module*. CORAL provides a convenient syntax for defining predicates using the extended version of C++, and for invoking queries on the declarative modules from the imperative module. Declarative modules are compiled into C++ classes, and the runtime system is written in C++. Thus, CORAL reduces the impedance mismatch between the declarative modules and the C++ module at both the language level and the implementation level.

While there are many similarities to the other deductive database systems mentioned earlier, there are important differences as well. For example, CORAL supports non-ground terms and ground multisets. The use of general updates is not allowed in declarative rules. A number of the control

[3] CORAL stands for "COmbining Relations And Logic".

features supported in CORAL are novel, as is the approach to integrating with an imperative language such as C++.

8.1. Declarative Modules

Declarative programming is supported by allowing the user to organize sets of rules and facts into *declarative modules*. We introduce the syntax using a program to append two lists.

module $Listroutines$.
export $append$ (bbf, bfb, fbb).

$append([\ ], L, L).$
$append([H \mid T], L, [H \mid L1]) :- append(T, L, L1).$

end_module.

This program illustrates the notion of modules, and CORAL's support for complex objects such as lists. The *export* statements at the beginning of each module define what forms of external queries are permitted on the module ("b" denotes an argument that must be bound in the query, and "f" an argument that can be free).

CORAL allows facts to have variables within them. Thus, it is possible to query *append* as follows:

Query: ?-$append([1, 2, 3, 4, X], [Y, Z], ANS)$.

and get the answer

$ANS = [1, 2, 3, 4, X, Y, Z]$

In this CORAL differs from Aditi, EKS-V1, LDL, Glue/NAIL! and Starburst SQL, which restrict the facts in a database to be ground. The cost and benefits of non-ground facts is discussed in Section 5.10.

8.1.1. Sets and Negation

The keyword not is used as a prefix to indicate a negated body literal. A program is non-floundering if all variables in a negated literal are ground before the literal is evaluated (in the left-to-right rule order). CORAL supports a class of programs with negation that properly contains the class of non-floundering modularly stratified programs (Ross [85]).

Sets can be constructed by *set-enumeration*, or by *set-grouping*. Set grouping was discussed in Section 6.2. The following rule (fact) illustrates set-enumeration.

$children(john, \{mary, peter, paul\}).$

All elements of sets generated thus must be ground terms (i.e., not contain any variables). General matching or unification of sets (where one or both of the sets can have variables, respectively) is not supported. CORAL requires the use of the set-grouping operator to be modularly-stratified (in the same way as negation). When programs allow the creation of sets (and nested structures that include sets) the domain (or universe) of discourse has to be extended beyond the Herbrand universe that is standard in logic programming. Beeri et al. [12] describe such an extended Herbrand universe.

CORAL provides several standard operations on sets and multisets as built-in predicates. These include member, union, intersection, difference, multisetunion, cardinality, subset, and set. CORAL also provides several aggregate operations for use on sets and multisets such as count, min, max, sum, product, average and any.

8.2. Other Declarative Module Features

CORAL uses bottom-up evaluation as the basic paradigm for evaluation of declarative modules. It uses a variety of optimization techniques including program transformations such as Magic Templates ([13; 74]), optimizations of bottom-up evaluation such as Semi-naive evaluation, rule and predicate orderings, efficient unification techniques such as hash-consing [36; 90] and sharing of structure for ground terms, and efficient indexing techniques. Currently, evaluation is completely in-memory, although we plan to implement disk-resident relations in future versions of CORAL. More optimization techniques will be incorporated in future versions of CORAL.

Modules are the unit of compilation in CORAL. Numerous annotations are allowed in modules, and many of these have a global effect on the module. Unlike Glue/NAIL!, where modules have only a compile time meaning and no run-time meaning, modules in CORAL have important run-time semantics. Several run-time optimizations are done at the module level.

In most cases, facts (other than answers to the query) computed during the evaluation of a module are best discarded to save space (since bottom-up evaluation stores many facts, space is generally at a premium). Module calls provide a convenient unit for discarding intermediate answers, which may take up too much space otherwise. By default CORAL does precisely this — it discards all intermediate facts and subgoals computed by a module at the end of a call to the module. This can be overridden by the user, by using an annotation save_module within a module definition.

CORAL provides utilities to take a text file organized as a table, parse it into fields and records, and convert it into a relation. Similarly, utilities for output of relations in tabular form are also provided. This provides some of the features of the AWK string manipulation language that is available on Unix systems.

CORAL supports Ordered Search [80], which is an evaluation mechanism that orders the use of generated subgoals in a program. Subgoals and answers to subgoals are generated asynchronously, as in the regular bottom-up evaluation of the magic program. However, the order in which generated subgoals are made available for use is somewhat similar to a top-down evaluation. Ordered search maintains information about dependencies between subgoals, which can be used to evaluate a large class of programs with negation. It also provides an ordering to the computation that hides subgoals; this could be useful in situations when only a single answer is desired for a query.

CORAL provides some basic facilities for debugging of programs. A trace facility lets the user know what rules are being evaluated, and prints out answers and subgoals as they are generated. CORAL also provides some high-level profiling facilities, that use as the unit of measurement the unification operation (unification of large subterms are treated as generating recursive calls to unification for the purpose of this count).

8.3. Imperative Modules in CORAL

An imperative module is a program in an imperative language, which consists of C++ augmented by adding a layer of new types and constructs. We introduce some features of imperative modules using the program in Figure 1, which updates the salary of a person depending on the number of employees that work for the person (directly or indirectly). The program gives an intuitive idea of the features we provide for imperative modules.

A *relation* is a set of tuples. The user can create two types of relations: *unindexed relations* and *indexed relations*. Indices can be added to an existing *indexed* relation by means of a procedure call.

A C++ user can also directly access a database relation (not just get a copy of it) by providing the name of the relation and its arity. CORAL provides facilities to insert tuples into, and delete tuples from relations using the "+ =" and "− =" operators. A procedure update_tuple to update tuples in a relation is also provided, as illustrated in the example.

CORAL provides two iterative constructs for accessing the tuples of a relation (the tuples are returned in an arbitrary order). Both these are illustrated in the example. FOR_EACH_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument. FOR_EACH_MATCHING_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument that matches the pattern specified by the third argument. A variety of other functions are available to the imperative language programmer to manipulate relations. These include all the set and aggregate functions described earlier.

A C++ user can invoke a query on a relation that is defined declaratively

```
void Update_Sals (Relation *emp)
{

    TupleType(EmpTuple,3)
        TupleArg(1, string, ename);    TupleArg(2, string, mname);
        TupleArg(3, int, sal);
    EndTupleType(EmpTuple)
    TupleType(NumempsTuple,2)
        TupleArg(1, string, mname);    TupleArg(2, int, numemps);
    EndTupleType(NumempsTuple)
    EmpTuple *emp_tuple, new_tuple;
    NumempsTuple query, *result_tuple, pattern;
    Relation *numofemps = new IndexedRelation(2);

    query.set_mname(make_var(0));
    query.set_numemps(make_var(1));
    call_coral ("numofemps", &query, numofemps);

    FOR_EACH_TUPLE (emp_tuple, emp) {
        pattern.set_mname( emp_tuple→ename());
        pattern.set_numemps( make_var(0) );
        int newsal = emp_tuple→sal();
        FOR_EACH_MATCHING_TUPLE(result_tuple, numofemps,
                                          &pattern) {
            newsal += 10* result_tuple→numemps();
        } END_EACH_TUPLE(result_tuple)
        copy_tuple(emp_tuple, new_tuple);
        new_tuple→set_sal(newsal);
        update_tuple(emp, emp_tuple, new_tuple);
    } END_EACH_TUPLE(emp_tuple)

}
```

module Employee.
export $numofemps$ (bf).

$$
\begin{aligned}
worksfor(E, M) &:- emp(E, M, S).\\
worksfor(E, M) &:- worksfor(E, E1),\\
&\qquad emp(E1, M, S)\\
numofemps(M, \mathsf{count}(set(< E >))) &:- worksfor(E, M).
\end{aligned}
$$

end module.

Fig. 1. Updating Employee Salaries

(and exported by a declarative module), using the procedure call_coral illustrated in the example. There are two variants of this procedure, one of which takes a single query, and the other a set of queries. In later versions of CORAL we plan to allow inline declaration of a declarative module within imperative code. This will provide a simpler syntax for calling declarative modules and could be interpreted as a direct extension of the C++ language. CORAL provides a simple convention for defining predicates using C++ code.

To provide efficient support for novel applications, CORAL allows the user to create new abstract data types and integrate them with the declarative query language in a simple and clean fashion, without modifying or recompiling system code. For instance, when dealing with DNA sequences, a type *sequence* that provides several built-in operations such as approximate subsequence matching, indexing etc. is very useful.

9. Conclusion

We have reviewed a collection of results on bottom-up evaluation of logic programs, and attempted to place them in the perspective of a coherent approach to logic program evaluation. The main points, in our opinion, are the following:

- Efficient bottom-up evaluation methods are available that are sound and complete with respect to the declarative semantics.
- Systems based upon these methods are being developed, and offer rich support for rule-based applications.

Acknowledgements

This paper is based upon a tutorial to be presented at COMPEURO 92 by R. Ramakrishnan. It has borrowed some material from other papers, in particular [64].

This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

References

1. Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, San Antonio, Texas, 1979.
2. P.M.G. Apers, M.A.W. Houtsma, and F. Brandse. Processing recursive queries in relational algebra. In *Data and Knowledge (DS-2), Proc. of the Second IFIP 2.6 Working Conference on Database Semantics*, pages 17–39. North Holland, 1986.

3. K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, San Mateo, Calif., 1988.
4. I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. Journal of Logic Programming. To Appear.
5. I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.
6. Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems*. Springer-Verlag, 1985.
7. Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.
8. Marianne Baudinet, Marc Niezette, and Pierre Wolper. On the representation of infinite temporal data and queries. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 280–290, Denver, Colorado, May 1991.
9. R. Bayer. Query evaluation and recursion in deductive database systems. Unpublished Memorandum, 1985.
10. C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Valid computations and the Magic implementation of stratified programs. Manuscript, September 91.
11. Catriel Beeri, Shamim Naqvi, Raghu Ramakrishnan, Oded Shmueli, and Shalom Tsur. Sets and negation in a logic database language. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 21–37, San Diego, California, March 1987.
12. Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *The Journal of Logic Programming*, pages 181–232, 1991.
13. Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.
14. R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Computational Logic*, pages 101–116, 1972.
15. F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Procs. International Conference on Extending Database Technology*, February 1988.
16. Francois Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312, 1990.
17. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.
18. Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, April 1985.
19. Weidong Chen, Michael Kifer, and Davis S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, 1989.
20. D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the International Conference on Very Large Databases*, pages 195–204, 1989.
21. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, 1990.
22. L. Cholvy and R. Demelombe. Querying a rule base. In *Procs. 1st International*

Conference on Expert Database Systems, pages 365–371, April 1986.
23. Jan Chomicki. Polynomial time query processing in temporal deductive databases. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 379–391, Nashville, Tennessee, April 1990.
24. S.R. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 200–216, Philadelphia, Pennsylvania, March 1989.
25. M. P. Consens and A. O. Mendelzon. Low complexity aggregation in Graphlog and Datalog. In *Procs. International Symposium on Database Theory*, Paris, 1990.
26. Stavros S. Cosmadakis, Haim Gaifman, Paris Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the Twentieth Symposium on the Theory of Computation*, pages 477–490, Chicago, Illinois, May 1988.
27. I. F. Cruz and T. S. Norvell. Aggregative closure: An extension of transitive closure. In *Proc. IEEE 5th Int'l Conf. Data Engineering*, pages 384–389, 1989.
28. Suzanne W. Dietrich and David S. Warren. Extension tables: Memo relations in logic programming. In *Proceedings of the Symposium on Logic Programming*, pages 264–272, 1987.
29. G. Dong. On distributed processing of Datalog queries by decomposing databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–35, Portland, Oregon, June 1986.
30. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, 19(1):17–37, 1982.
31. Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 106–115, Ithaca, New York, June 1987.
32. S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of Datalog queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990. To appear.
33. Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.
34. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.
35. R Gonzalez-Rubio, J. Rohmer, and A. Bradier. An overview of DDC: Delta Driven Computer. In *Parallel Architectures and Languages Europe, Volume 1: Parallel Architectures*, Lecture Notes in Computer Science, No. 258, pages 414–433, 1987.
36. E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan, May 1974.
37. A. Richard Helm. Detecting and eliminating redundant derivations in deductive database systems. Technical Report RC 14244 (#63767), IBM Thomas Watson Research Center, December 1988.
38. Yannis E. Ioannidis. Bounded recursion in deductive databases. *Algorithmica*, 1(4):361–385, October 1986.
39. Yannis E. Ioannidis and Eugene Wong. Towards an algebraic theory of recursion. Technical Report 801, Computer Sciences Department, University of Wisconsin-Madison, October 1988.
40. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL*, pages 111–119, Munich, January 1987.
41. Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.
42. D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule

transformations that maintain context information. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 380–391, August 1990.
43. David Kemp, Divesh Srivastava, and Peter Stuckey. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the International Logic Programming Symposium*, pages 337–351, San Diego, CA, U.S.A., October 1991.
44. David Kemp and Peter Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., October 1991.
45. J.M. Kerisit and J.M. Pugin. Efficient query answering on stratified databases. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 719–725, Tokyo, Japan, November 1988.
46. Michael Kifer and Ai Li. On the semantics of rule based expert systems with uncertainty. In *Proceedings of 2nd ICDT*, Bruges, Belgium, 1988.
47. Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive databases. In *Proceedings of the Advanced Database Symposium*, Tokyo, Japan, 1986.
48. Michael Kifer and Eliezer L. Lozinskii. SYGRAF: Implementing logic programs in a database style. *IEEE Transactions on Software Engineering*, 1988.
49. J. J. King. QUIST: A system for semantic query optimization in relational databases. In *Proceedings of the International Conference on Very Large Databases*, pages 510–517, August 1981.
50. J. Kuittinen, O. Nurmi, S. Sippu, and E. Soisalon-Soininen. Efficient implementation of loops in bottom-up evaluation of logic queries. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
51. Michael J. Maher. *Semantics of Logic Programs.* PhD thesis, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1985.
52. Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990.
53. Katherine Morris, Jeffrey F. Naughton, Yatin Saraiya, Jeffrey D. Ullman, and Allen Van Gelder. YAWN! (Yet Another Window on NAIL!). *Database Engineering*, December 1987.
54. Katherine Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming*, 1986.
55. A. Motro. SEAVE: The mechanism for verifying user presuppositions. *ACM TOIS*, 4(4):312–330, October 1986.
56. I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, New Jersey, May 1990.
57. Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
58. Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 314–330, Nashville, Tennessee, April 1990.
59. S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1988.
60. Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.
61. Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, pages 312–319, Chicago, Illinois, May 1988.
62. Jeffrey F. Naughton. Data independent recursion in deductive databases. *Journal of Computer and System Sciences*, 38(2):259–289, April 1989.
63. Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without

repeating it. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

64. Jeffrey F. Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In J-L. Lassez, editor, *Computational Logic: Essays in Honor of Alan Robinson*. The MIT Press, 1991.
65. Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.
66. Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–242, Portland, Oregon, May 1989.
67. Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 227–236, San Diego, California, March 1987.
68. Jeffrey F. Naughton and Yehoshua Sagiv. Minimizing expansions of recursions. In Hasan Ait-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 1, pages 321–349, San Diego, California, 1989. Academic Press, Inc.
69. F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the twenty-first Annual Meeting of the Association for Computational Linguistics*, 1983.
70. Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 308–317, 1991.
71. H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, 1988.
72. T.C. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.
73. T.C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Seventh International Logic Programming Conference*, pages 459–477, 1990.
74. Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.
75. Raghu Ramakrishnan. Parallelism in logic programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990.
76. Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential Datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.
77. Raghu Ramakrishnan, Per Bothner, Divesh Srivastava, and S. Sudarshan. Coral: A database programming language. In Jan Chomicki, editor, *Proceedings of the NACLP '90 Workshop on Deductive Databases*, October 1990. Available as Report TR-CS-90-14, Department of Computing and Information Sciences, Kansas State University.
78. Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Vardi. Prooftree transformation theorems and their applications. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, March 1989.
79. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.
80. Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search

in bottom-up evaluation. Manuscript, submitted for publication, 1991.
81. Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.
82. Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In preparation (full version of paper that appeared in ILPS'91), 1992.
83. Peter Z. Revesz. A closed form for datalog queries with integer order. In *International Conference on Database Theory*, pages 187–201, France, December 1990.
84. J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.
85. Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.
86. Kenneth Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.
87. Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the post-ILPS'91 Workshop on Deductive Databases*, 1991.
88. Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.
89. Yatin Saraiya. Linearizing nonlinear recursions in polynomial time. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 182–189, Philadelphia, Pennsylvania, March 1989.
90. M. Sassa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(4):31–34, June 1976.
91. Helmut Schmidt. *Meta-Level Control for Deductive Database Systems.* Lecture Notes in Computer Science, Number 479. Springer-Verlag, 1991.
92. H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.
93. T. Sellis, C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 404–412, June 1988.
94. Ehud Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proceedings of IJCAI*, pages 529–532, 1983.
95. Seppo Sippu and Eljas Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.
96. S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, September 1991.
97. S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In preparation, 1992.
98. S. Sudarshan, Divesh Srivastava, Raghu Ramakrishnan, and Jeff Naughton. Space optimization in the bottom-up evaluation of logic programs. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, 1991.
99. Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(4):289–321, September 1985.
100. Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, March 1989.
101. Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, and Peter Stuckey. The Aditi deductive database system. In *Proceedings of the NACLP'90 Workshop on Deductive Database Systems*, 1990.
102. M. H. Van Emden. Quantitative deduction and its fipoint theory. *Journal of Logic Programming*, (1):37–53, 1986.

103. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.
104. A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
105. Allen Van Gelder. A message passing framework for logical query evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 349–362, Washington, DC, May 1986.
106. Moshe Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 341–351, Austin, Texas, March 1988.
107. L. Vieille, P. Bayer, and V. Küchenhoff. Integrity checking and materialized views handling by update propagation in the EKS-V1 system. Technical report, CERMICS - Ecole Nationale Des Ponts Et Chaussees, France, June 1991. Papport de Recherche, CERMICS 91.1.
108. L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, 1990.
109. Laurent Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proceedings of the First International Conference on Expert Database Systems*, pages 179–193, Charleston, South Carolina, 1986.
110. Laurent Vieille. Database complete proof procedures based on SLD-resolution. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 74–103, 1987.
111. Laurent Vieille. From QSQ towards QoSaQ: Global optimizations of recursive queries. In *Proc. 2nd International Conference on Expert Database Systems*, April 1988.
112. J. Widom, R.J. Cochrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the International Conference on Very Large Databases*, pages 275–285, August 1991.
113. O. Wolfson and A. Silberschatz. Sharing the load of logic program evaluations. In *Proceedings of the 7th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Philadelphia, Pennsylvania, March 1989.
114. W. Zhang, C. T. Yu, and D. Troy. A necessary and sufficient condition to linearize doubly recursive programs in logic databases. Unpublished manuscript, Department of EECS, University of Illinois at Chicago, 1988.